Compute Ontario Colloquium

Parallel Programming: MPI I/O Basics

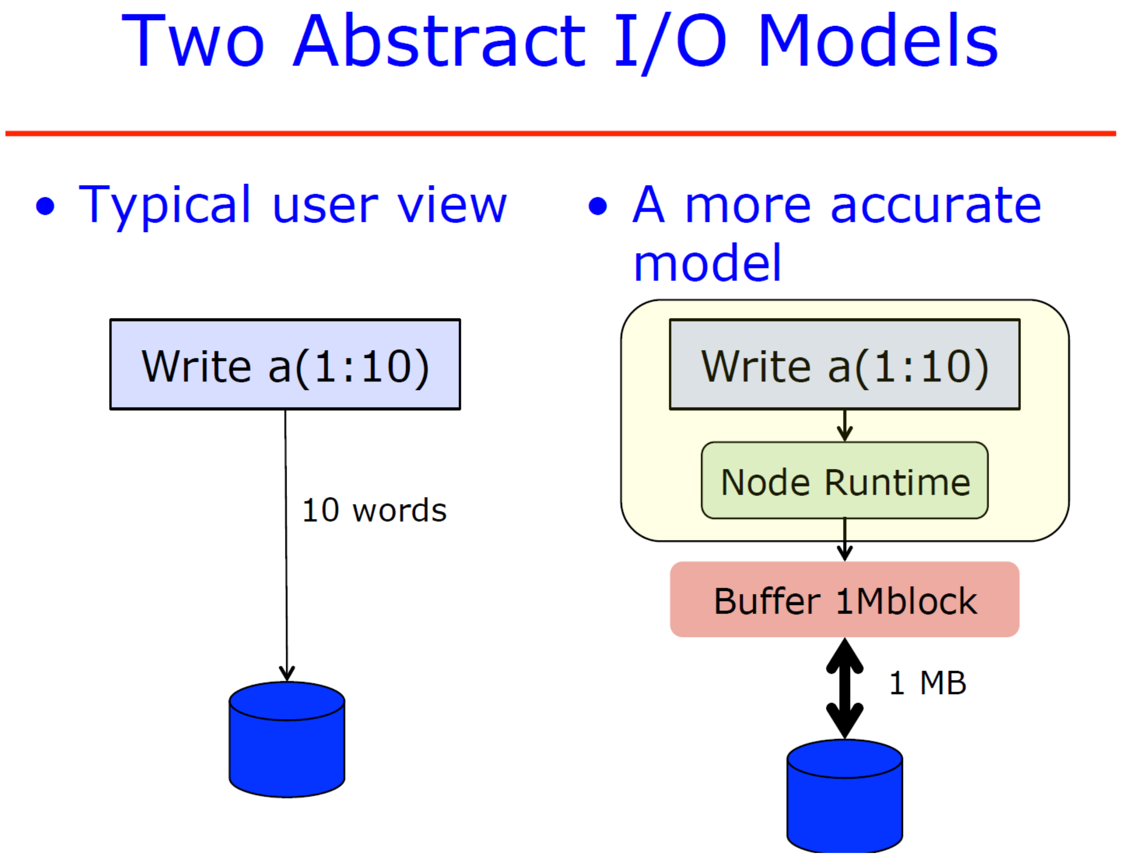
Jemmy Hu

SHARCNET HPC Consultant

Oct. 23, 2024

I/O Basics

- I/O (input/output, read/write) is needed in all programs but is often overlooked
- Mapping problem: how to convert internal structures and domains to files which are a streams of bytes
- Transport problem: how to get the data efficiently from hundreds of nodes on the supercomputer to physical disks
- File system: size, location
- File handle: open, read, write, view, close, etc..
- Speed/performance: size, scalability



Two Abstract I/O Models

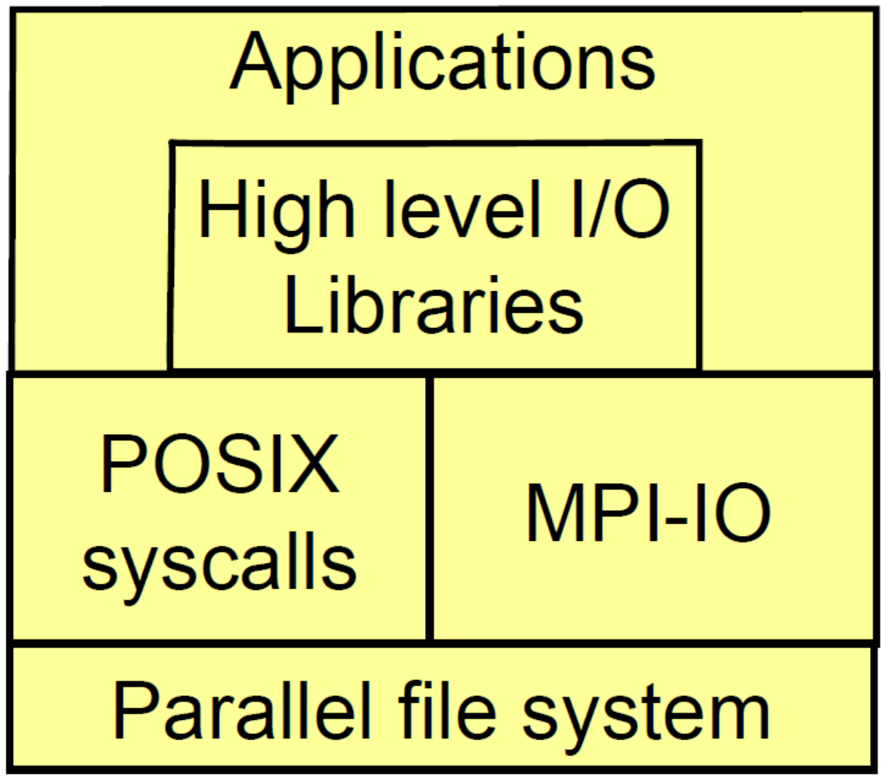
- Typical user view
- A more accurate model

Parallel I/O

- Deal with very large datasets while running massively parallel applications on supercomputers
- Good I/O is non-trivial, the challenges are
  - – performance, scalability, reliability – Ease of use of output (number of files, format)
  - – amount of data saved is increased, latency to access to disks is not negligible
  - – data portability
- One cannot achieve all of the above - Solutions to managing IO in parallel applications must take into account different aspects of the application and implementation, one needs to decide what is most important
- **At the program level:**
  - Concurrent reads or writes from multiple processes to a common file
- **At the system level:**
  - A parallel file system and hardware that support such concurrent access

IO Layers

- High-level
  - application: to read or write data from disk
- Intermediate-level
  - high-level libraries
    HDF5, NETCDF
  - libraries or system tools for I/O
- Low-level
  - parallel filesystem enables the actual parallel I/O
  - Lustre, GPFS, PVFS



Ways to organize I/O

- Should be considered in the context of the entire application workflow, not just one program

- Several natural choices
  - One file per program
    - May match workflow, other tools
  - One file per process
    - Avoids performance and correctness bugs in the File system
  - One file per node/row/rack/…

Managing IO: Basic serial I/O (Single writer)

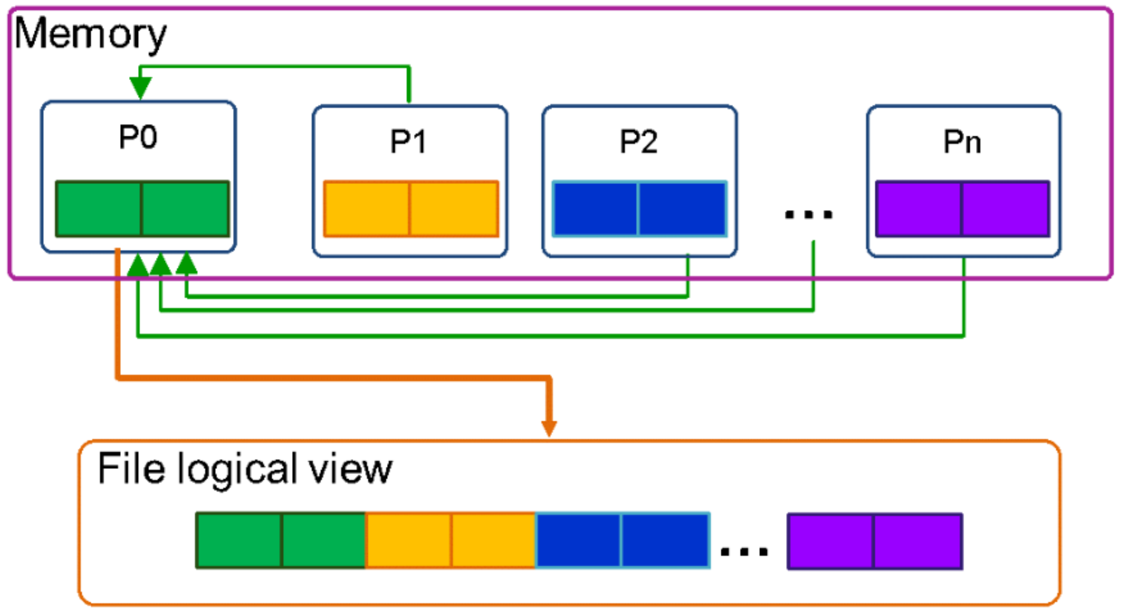
- One process performs I/O.
  - Data Aggregation or Duplication
  - Limited by single I/O process.

- Easy to program

- Pattern does not scale.
  - Time increases linearly with amount of data.
  - Time increases with number of processes.

- Care has to be taken when doing the “all to one”- kind of communication at scale

- Can be used for a dedicated IO Server (not easy to program) for small amount of data

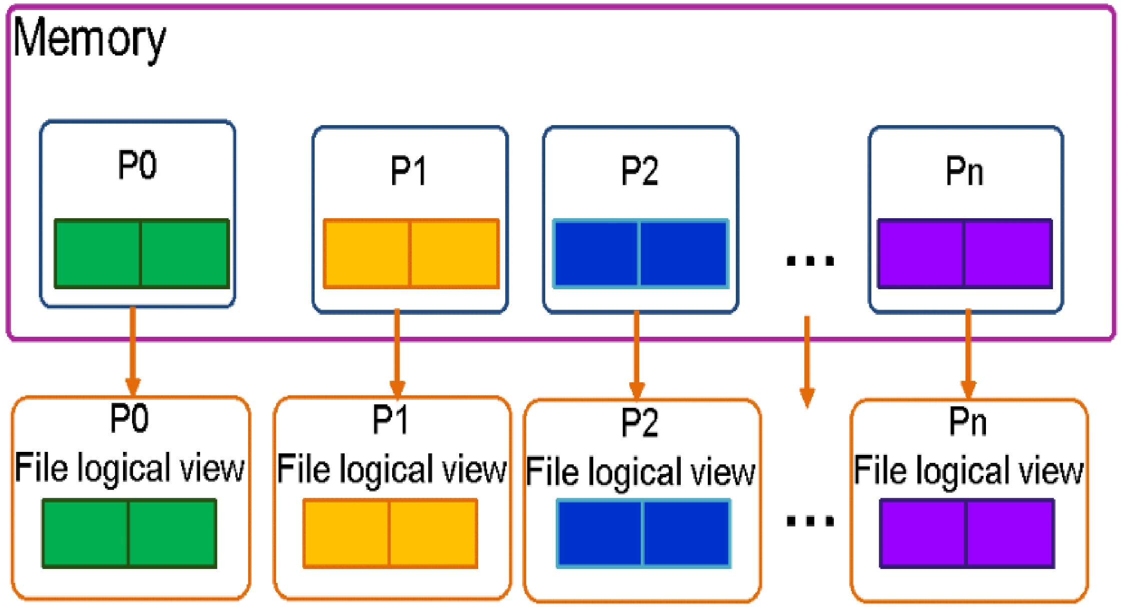


Managing IO: distributed IO (One file per process)

- All processes perform I/O to individual files.
  - Limited by file system.

- Easy to program

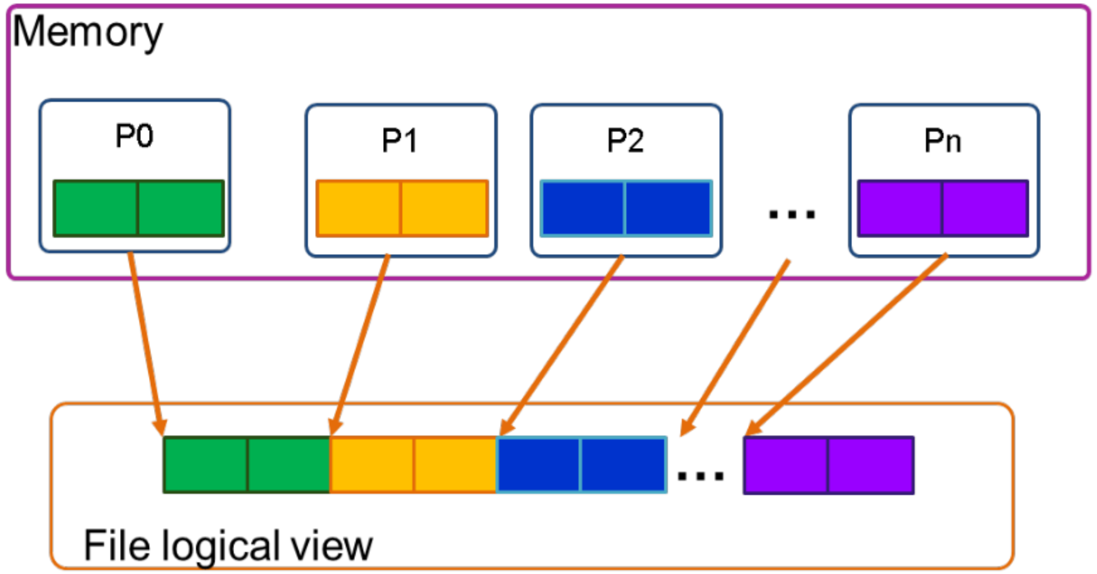
- Pattern does not scale at large process counts.
  - Number of files creates bottleneck with metadata operations.
  - Number of simultaneous disk accesses creates contention for file system resources.



Managing IO: Shared File (Independent writers)

- All processes write to specific blocks of the file, but synchronization is necessary to prevent write conflicts.
- Coordination usually occurs at the parallel file system level.
  - Although communication between processes is not required, multiple processes may compete for certain regions of the file, leading to lock contention, hence lower performance.

- High-level I/O libraries such as parallel HDF5, pNetCDF, and ADIOS2 are commonly used with this approach and help to encapsulate data within the file, but the complexity of managing file regions is not visible to the programmer.



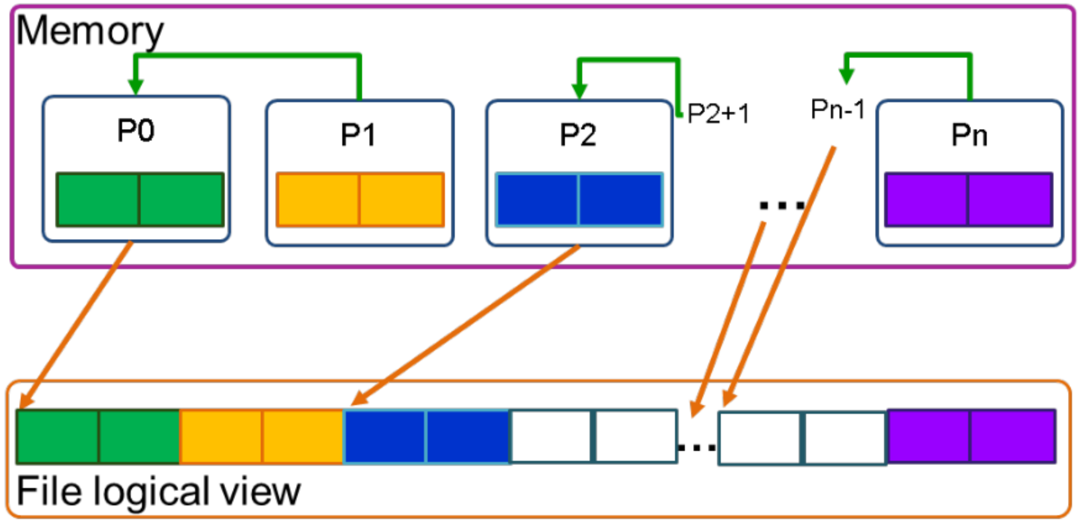
Managing IO: Shared File (Collective writers)

- Improve performance in the single file approach

- Data aggregation can be done by introducing *aggregators*

- Chosen processes that collect data from others and write it to specific sections of the file. However, this method may be complex to implement.

- Using a high-level library can make the process of data aggregation and file management much easier, as it typically handles these tasks behind the scenes and allows for a single logical view of the file while also providing control over the aggregation strategy.



MPI I/O

MPI I/O was introduced in MPI-2

- A set of extensions to the MPI library that enable parallel high-performance I/O operations
- Provides a parallel file access interface that allows multiple processes to write and read to the same file simultaneously

- Defines parallel operations for reading and writing files:
  - I/O to only one file and/or to many files
  - Contiguous and non-contiguous I/O
  - Individual and collective I/O
  - Asynchronous I/O

- Portable programming interface
- Efficient data transfer between processes, Potentially good performance
  - Enables high-performance I/O operations on large datasets
  - Used as the backbone of many parallel I/O libraries such as parallel NetCDF and parallel HDF5

Basic Concepts in MPI I/O

- File *handle*
  - –data structure which is used for accessing the file

- File *pointer*
  - –*position*ing the file where to read or write
  - –can be individual for all processes or shared between the processes
  - –accessed through file handle

Basic Concepts in MPI I/O

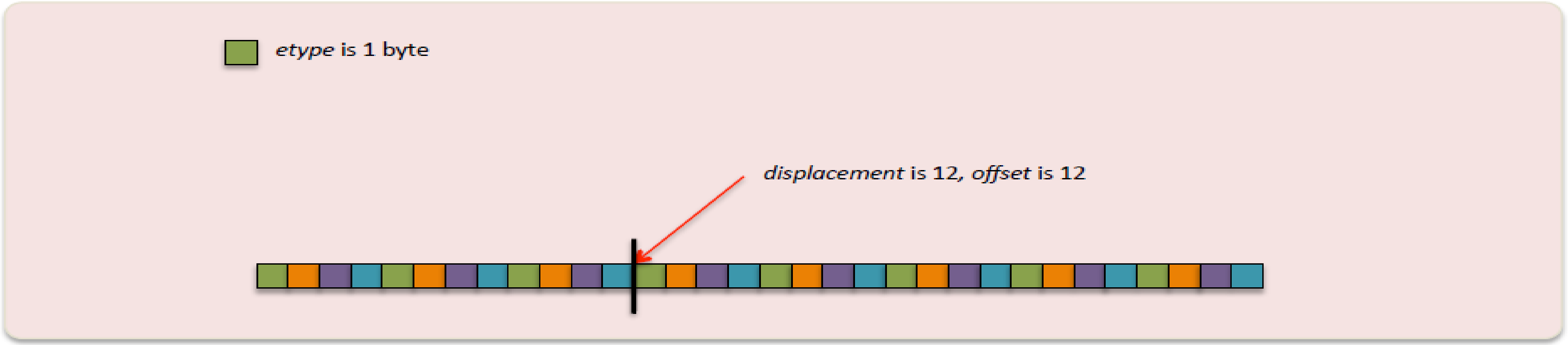
- File *view*
  - –part of a parallel file which is visible to process
  - –enables efficient non-contiguous access to file

- *Collective* and *independent* I/O
  - –collective = MPI coordinates the reads and writes of processes
  - –independent = no coordination by MPI

Basic Concepts in MPI I/O

- **Displacements**
  - The displacement of a position within a file is the number of bytes from the beginning of the file
- **Elementary Datatype**
  - An etype is an MPI datatype (predefined or a derived datatype)
  - The *etype* is used to set file views and for file access operations (reads and writes)
- **Offset**

  An offset is a position in the file given in terms of multiples of *etypes*
  - It is a multiple of *etypes* from the beginning of the current *view*
    - On file open the *view* begins at the start of the file
    - On file open the *etype* is a byte



Basic MPI-IO Operations

- MPI-IO provides basic IO operations:
  - open, seek, read, write, close (etc.)
- open/close are collective operations on the same file
  - many modalities to access the file (composable: |,+)
- read/write are similar to send/recv of data to/from a buffer
  - each MPI process has its own local pointer to the file (individual file pointer) for seek, read, write operations
  - offset variable is a particular kind of variable and it is given in elementary unit (etype) of access to file (default in byte)
  - it is possible to know the exit status of each subroutine/function

Opening a file

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)
```

```
MPI_File_open(comm, filename, amode, info, fh, ierr)

Character(*) :: filename
Integer :: comm, amode, info, fh, ierr
```

- A collective call for all processes in a communicator to open a file
- comm: communicator that performs parallel I/O, typically use MPI_COMM_WORLD
- amode: file access mode, MPI_MODE_RDONLY, MPI_MODE_WRONLY, MPI_MODE_CREATE, MPI_MODE_RDWR, …
- Info: Hints to implementation for optimal performance (No hints: MPI_INFO_NULL)
- fh: parallel file handle

File Access Modes (amode)

A number of access modes are supported for MPI files
- The amode argument to MPI_File_open defines the access mode for the file

Multiple access modes can be combined by
- Using addition or the IOR function in Fortran
  - i.e. MPI_MODE_CREATE+MPI_MODE_EXCL+MPI_MODE_WRONLY
- Using the or (|) operator in C
  - i.e. MPI_MODE_CREATE|MPI_MODE_EXCL|MPI_MODE_WRONLY

MPI_MODE_RDONLY	open for read only
MPI_MODE_RDWR	open for reading and writing
MPI_MODE_WRONLY	open for write only
MPI_MODE_CREATE	create the file if it does not exist
MPI_MODE_EXCL	generate an error if creating a file that already exists
MPI_MODE_DELETE_ON_CLOSE	delete the file on MPI_File_close is called
MPI_MODE_APPEND	set initial position of all file pointers to end of file
MPI_MODE_UNIQUE_OPEN	
MPI_MODE_SEQUENTIAL	

Closing a file

```
int MPI_File_close(MPI_File *fh)
```

```
MPI_File_close(fh, ierr)

Integer :: fh, ierr
```

- Collective operation, the same communicator use to open the file
- Call this function when the file access is finished to free the file handle.

File seek

File pointer

- Position in the file where to read or write
- Can be individual for all processes or shared between the processes
- Each process moves its local file pointer (individual file pointer) with

MPI_File_seek(fh, disp, whence)

fh: file handle, data structure which is used for accessing the file

disp: Displacement in bytes (with default file view)

whence: MPI_SEEK_SET: the pointer is set to offset

MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset

MPI_SEEK_END: the pointer is set to the end of the file plus offset

File reading

- Read file at individual file pointer

MPI_File_read (fhandle, buf, count, datatype, status)

buf: Buffer in memory where to read the data

count: number of elements to read

datatype: datatype of elements to read

status: similar to status in MPI_Recv, amount of data read can be determined by MPI_Get_count

– Updates position of file pointer after reading

– Not thread safe

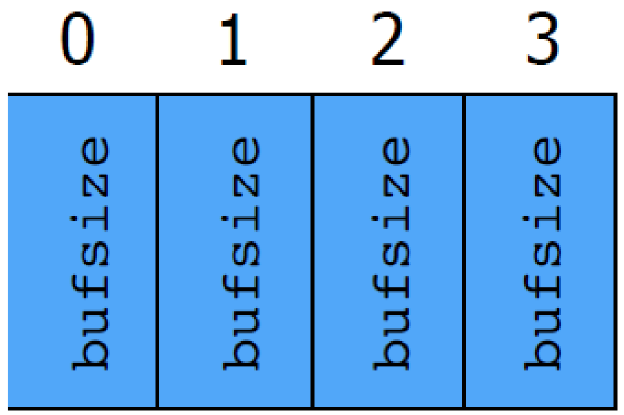
Parallel read: example in C

```
#include "mpi.h"

int main(int argc, char **argv) {
    int rank, nprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_File fh;
    MPI_Status status;
    MPI_File_open(MPI_COMM_WORLD, "../datafile", MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

    MPI_Offset filesize;
    MPI_File_get_size(fh, &filesize);
    MPI_Offset bufsize = filesize/nprocs;
    int nints = bufsize/sizeof(int);
    int *buf = (int*) malloc(nints);
   MPI_File_seek(fh, rank*bufsize, MPI_SEEK_SET);
   MPI_File_read(fh, buf, nints, MPI_INT, &status);
   MPI_File_close(&fh);
   MPI_Finalize();
}
```



File offset determined by MPI_File_seek

File writing

- Similar to reading
  - File opened with MPI_MODE_WRONLY or MPI_MODE_CREATE

- Write file at individual file pointer

  MPI_File_write (fhandle, buf, count, datatype, status)

  - Updates position of file pointer after writing
  - Not thread safe

Parallel write: example in Fortran

Multiple processes write to a binary file ‘test’. First process writes integers 1-100 to the beginning of the file, etc.

```
program output
  use mpi
  implicit none
  integer :: err, i, myid, file, intsize
  integer :: status(mpi_status_size)
  integer, parameter :: count=100 integer, dimension(count) :: buf
  integer(kind=mpi_offset_kind) :: disp
  call mpi_init(err)
  call mpi_comm_rank(mpi_comm_world, myid, err)
  do i = 1, count
    buf(i) = myid * count + i
  end do
  call mpi_file_open(mpi_comm_world, 'test', mpi_mode_wronly + mpi_mode_create, &
  mpi_info_null, file, err)
  call mpi_type_size(mpi_integer, intsize,err)
  disp = myid * count * intsize
  call mpi_file_seek(file, disp, mpi_seek_set, err)
  call mpi_file_write(file, buf, count, mpi_integer, status, err)
  call mpi_file_close(file, err)
  call mpi_finalize(err)
end program output
```

File offset determined by MPI_File_seek

File reading, explicit offset

- The location to read or write can be determined also explicitly with

MPI_File_read_at (fhandle, disp, buf, count, datatype, status)

disp: displacement in bytes (with the default file view) from the beginning of file

–Thread-safe

–The file pointer is neither referred or incremented

File writing, explicit offset

- Determine location within the write statement (explicit offset)

MPI_File_write_at (fhandle, disp, buf, count, datatype, status)

  - Thread-safe
  - The file pointer is neither used or incremented

Parallel write: explicit offset

```
program output
  use mpi
  implicit none
  integer :: err, i, myid, file, intsize
  integer :: status(mpi_status_size)
  integer, parameter :: count=100 integer, dimension(count) :: buf
  integer(kind=mpi_offset_kind) :: disp
  call mpi_init(err)
  call mpi_comm_rank(mpi_comm_world, myid, err)
  do i = 1, count
    buf(i) = myid * count + i
  end do
  call mpi_file_open(mpi_comm_world, 'test', mpi_mode_wronly + mpi_mode_create, &
  mpi_info_null, file, err)
  call mpi_type_size(mpi_integer, intsize,err)
  disp = myid * count * intsize

  call mpi_file_write_at(file, disp, buf, count, mpi_integer, status, err)
  call mpi_file_close(file, err)
  call mpi_finalize(err)
end program output
```

Multiple processes write to a binary file 'test'. First process writes integers 1-100 to the beginning of the file, etc.

File offset determined explicitly

Parallel read with explicit offset: example in Fortran

Note: The first part is the same as in the example write code shown on last slide.
Same number of processes for reading and writing is assumed in this example.

File offset determined explicitly

```
...
  call mpi_file_open(mpi_comm_world, 'test’, &
        mpi_mode_rdonly, mpi_info_null, file, err)

  intsize= sizeof(count)
  disp=myid * count * Intsize

  call mpi_file_read_at(file, disp, buf, count, mpi_integer, status, err)
  call mpi_file_close(file, err)
  call mpi_finalize(err)

End program output
```

Collective operations

- I/O can be performed *collectively* by all processes in a communicator
  - MPI_File_read**_all**
  - MPI_File_write**_all**
  - MPI_File_read_at**_all**
  - MPI_File_write_at**_all**
- Same parameters as in independent I/O functions
  - MPI_File_read
  - MPI_File_write
  - MPI_File_read_at
  - MPI_File_write_at

Collective operations

- All processes in communicator that opened file must call function
- Performance potentially better than for individual functions
  - Even if each processor reads a non-contiguous segment, in total the read is contiguous

Non-blocking IO

- Independent, nonblocking IO

  This is just like non blocking communication.

  Same parameters as in blocking IO functions (MPI_File_read etc)

  - MPI_File_iread
  - MPI_File_iwrite
  - MPI_File_iread_at
  - MPI_File_iwrite_at
  - MPI_File_iread_shared
  - MPI_File_iwrite_shared

- Collective, nonblocking IO

C interfaces to MPI I/O routines

```
int MPI_File_open(MPI_Commcomm, char *filename, int amode, MPI_Infoinfo, MPI_File*fh)
int MPI_File_close(MPI_File*fh)
int MPI_File_seek(MPI_Filefh, MPI_Offsetoffset, int whence)
int MPI_File_read(MPI_Filefh, void *buf, int count, MPI_Datatypedatatype,
MPI_Status*status)
int MPI_File_read_at(MPI_Filefh, MPI_Offsetoffset, void *buf, int count,
MPI_Datatypedatatype, MPI_Status*status)
int MPI_File_write(MPI_Filefh, void *buf, int count, MPI_Datatypedatatype,
MPI_Status*status)
int MPI_File_write_at(MPI_Filefh, MPI_Offsetoffset, void *buf, int count,
MPI_Datatypedatatype, MPI_Status*status)

int MPI_File_set_view(MPI_Filefh, MPI_Offsetdisp, MPI_Datatypeetype, MPI_Datatypefiletype,
char *datarep, MPI_Infoinfo)
int MPI_File_read_all(MPI_Filefh, void *buf, int count, MPI_Datatypedatatype,
MPI_Status*status)
int MPI_File_read_at_all(MPI_Filefh, MPI_Offsetoffset, void *buf, int count,
MPI_Datatypedatatype, MPI_Status*status)
int MPI_File_write_all(MPI_Filefh, void *buf, int count, MPI_Datatypedatatype,
MPI_Status*status)
int MPI_File_write_at_all(MPI_Filefh, MPI_Offsetoffset, void *buf, int count,
MPI_Datatypedatatype, MPI_Status*status)
```

Fortran interfaces for MPI I/O routines

```
mpi_file_open(comm, filename, amode, info, fh, ierr)integer :: comm, amode, info, fh,
ierrcharacter* :: filename
mpi_file_close(fh, ierr)
mpi_file_seek(fh, offset, whence)
integer :: fh, offset, whence
mpi_file_read(fh, buf, count, datatype, status)integer :: fh, buf, count, datatype,
status(mpi_status_size)
mpi_file_read_at(fh, offset, buf, count, datatype, status)integer :: fh, offset, buf, count,
datatypeinteger, dimension(mpi_status_size) :: status
mpi_file_write(fh, buf, count, datatype, status)
mpi_file_write_at(fh, offset, buf, count, datatype, status)

mpi_file_set_view(fh, disp, etype, filetype, datarep, info)integer :: fh, disp, etype,
filetype, infocharacter* :: datarep
mpi_file_read_all(fh, buf, count, datatype, status)
mpi_file_read_at_all(fh, offset, buf, count, datatype, status)
mpi_file_write_all(fh, buf, count, datatype, status)
mpi_file_write_at_all(fh, offset, buf, count, datatype, status)
```

References

- https://www.nhr.kit.edu/userdocs/horeka/parallel_IO/
- https://hpc-forge.cineca.it/files/CoursesDev/public/2017/Parallel_IO_and_management_of_large_scientific_data/Roma/MPI-IO_2017.pdf
- https://janth.home.xs4all.nl/MPIcourse/PDF/08_MPI_IO.pdf
- https://events.prace-ri.eu/event/176/contributions/59/attachments/170/326/Advanced_MPI_II.pdf
- https://www.cscs.ch/fileadmin/user_upload/contents_publications/tutorials/fast_parallel_IO/MPI-IO_NS.pdf